In Search of a Fast and Efficient Serverless DAG Engine

Benjamin Carver, Jingyuan Zhang, Ao Wang, Yue Cheng

George Mason University

Abstract—Python-written data analytics applications can be modeled as and compiled into a directed acyclic graph (DAG) based workflow, where the nodes are fine-grained tasks and the edges are task dependencies.Such analytics workflow jobs are increasingly characterized by short, fine-grained tasks with large fan-outs. These characteristics make them well-suited for a new cloud computing model called serverless computing or Function-as-a-Service (FaaS), which has become prevalent in recent years. The auto-scaling property of serverless computing platforms accommodates short tasks and bursty workloads, while the pay-per-use billing model of serverless computing providers keeps the cost of short tasks low.

In this paper, we thoroughly investigate the problem space of DAG scheduling in serverless computing. We identify and evaluate a set of techniques to make DAG schedulers serverless-aware. These techniques have been implemented in WUKONG, a serverless, DAG scheduler attuned to AWS Lambda. WUKONG provides decentralized scheduling through a combination of static and dynamic scheduling. We present the results of an empirical study in which WUKONG is applied to a range of microbenchmark and real-world DAG applications. Results demonstrate the efficacy of WUKONG in minimizing the performance overhead introduced by AWS Lambda — WUKONG achieves competitive performance compared to a serverful DAG scheduler, while improving the performance of real-world DAG jobs by as much as $3.1\times$ at larger scale.

I. INTRODUCTION

In recent years, a new cloud computing model called serverless computing [8, 19] (Function-as-a-Service or FaaS)[1] has become prevalent, owing to OS-level (i.e., container-based) virtualization. Serverless computing enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions[2]; developers can thus focus on developing function logic without having to worry about provisioning, scaling, and managing traditional backend servers or VMs, which are notoriously tedious to maintain [16].

With their growth in popularity, serverless computing solutions have found their way into both commercial clouds (e.g., AWS Lambda, Google Cloud Functions, and IBM Cloud Functions) and open source projects (e.g., Open-Lambda [20]). While serverless platforms were originally intended for event-driven, stateless applications [1], recent trend has demonstrated the usage of serverless computing in support of more complex applications.

One such example is DAG (directed acyclic graph) workflow-based data analytics applications. These applications are characterized by short, fine-grained tasks with large fan-outs [27, 2, 17]. For example, an analysis of Alibaba workload traces shows that more than 50% of the analytics batch jobs, tasks, and instances are finished within 10 seconds [17, 12]. The auto-scaling property of serverless platforms makes these platforms well-suited for the short, fine-grained tasks and bursty, large fan-outs that characterize DAG-based workflows. In addition, FaaS providers charge users at a fine granularity – AWS Lambda bills on a per-invocation basis (\$0.02 per 1 million invocations) and charges resource usage by rounding up the function's execution time to the nearest 100 milliseconds (ms). Workloads with short tasks can take advantage of this fine-grained pay-as-you-go pricing model to keep monetary costs low[3]. Consequently, serverless computing can be leveraged as a promising solution for next-generation large-scale DAG workloads in high-performance computing (HPC), data analytics, and data sciences.

Moving DAG scheduling from a traditional serverful deployment to the emerging serverless platforms presents unique opportunities. In a traditional serverful deployment, the best practice is to utilize a logically centralized scheduler for managing task assignments and resource allocation under various objectives including load balancing, cluster utilization, fairness and so on. State-of-the-art serverful workflow schedulers include but are not limited to: MapReduce job scheduler [13], Apache Spark scheduler [31], Sparrow [27], and Dask [4]. In the context of serverless computing, however, the assumptions of the traditional serverful schedulers do not hold any more. This is because: (1) FaaS providers are responsible for managing the "servers" (i.e., where the task executors are hosted); and (2) serverless platforms typically provide a *nearly unbounded* amount of *ephemeral* resources. As a result, a hypothetical serverless DAG scheduler may not necessarily care about traditional "scheduling"-related metrics and constraints (such as load balancing and cluster utilization), as an individual task could be executed anywhere in the serverless data center that is essentially managed by the service provider.

Yet, designing a fast and efficient serverless-oriented DAG engine introduces challenges. First, a task needs to

---

[1]We use the term "serverless computing" and "FaaS" interchangeably.

[2]Without loss of generality, we use "Lambda functions" to represent cloud functions throughout.

[3]Pay-as-you-go in the context of serverless computing is essentially not paying for what are not being used.

be *dispatched* to a Lambda function as fast as possible. With this in mind, a logically centralized scheduler would inevitably introduce a performance bottleneck, especially for short-task dominated workloads. Second, as already mentioned, serverless platforms come with inherent constraints including limited outbound-only network connectivity; as such, a workflow has to rely on an external storage system for storing intermediate data, which impacts data locality and incurs extra network communications. Researchers have developed solutions on serverless computing platforms for supporting parallel jobs [21, 15, 29]; however, these attempts do not fully investigate decentralized DAG scheduling for serverless computing. Therefore, current state-of-the-art demands a new serverless-native DAG framework optimized to minimize the network communication overhead while maximizing data locality whenever possible.

In this paper, we argue that a serverless DAG engine urgently demands a radical redesign, with a focus shifted away from the techniques optimized for traditional DAG schedulers targeting serverful deployments. To this end, we present WUKONG, a serverless-oriented, decentralized, data-locality aware DAG engine. WUKONG uniquely exploits the elasticity of the serverless platform (in our case AWS Lambda) and completely delegates the requirements of load balancing, fairness, and resource efficiency to the serverless platform. WUKONG is novel in that it realizes *decentralized scheduling* where a DAG is partitioned into sub-graphs that are distributed to separate task executors (implemented as an AWS Lambda runtime). Lambda runtimes schedule the tasks in their sub-graphs and cooperate (at "joins" on sub-graph boundaries) to dynamically schedule tasks that are in two or more sub-graphs but that must only be executed once. WUKONG also provides efficient storage mechanisms for managing intermediate data; however, according to our factor analysis in §V-B, WUKONG's decentralized design has the most influence on its performance. The decentralized design minimizes communication overhead and improves scalability by increasing data locality.

Specifically, we make the following contributions:
- We thoroughly investigate the problem space of DAG scheduling in serverless computing,
- We identify and evaluate a set of techniques to make DAG scheduling serverless-aware,
- We design and implement WUKONG, a serverless DAG engine attuned to AWS Lambda,
- We evaluate WUKONG to validate its efficacy and design tradeoffs.

II. BACKGROUND AND RELATED WORK

A. Serverless Computing Primer

Serverless Computing handles virtually all the system administration operations to make it easier for users and developers to use a near-infinite amount of cloud resources including bundled CPUs and memory, object stores, and a lot more [22]. Service providers provide a flexible function interface so that developers can completely focus on development of the core application logic; service providers in turn help automatically scale the function executions in a demand-driven fashion, hiding the tedious cluster configuration and management overheads from the users.

General Constraints and Limitations: Service providers place limits on the use of cloud resources to simplify resource management. Take AWS Lambda for example: users have the flexibility of configuring Lambda's memory and CPU resources in a bundle. Users can choose a memory amount between 128MB and 3008MB in 64MB increments. Lambda allocates CPU power linearly in proportion to the amount of memory configured. Each Lambda function can run at most 900 seconds and will be forcibly returnedwhen function timeouts. In addition, Lambda only allows outbound TCP network connections and bans inbound connections and UDP protocol.

In addition to these constraints, serverless computing suffers from a "cold start" penalty [30, 26, 10][4] associated with container startups. Service providers rely on container caching (i.e., warmed functions) to mitigate the impact of cold starts on elasticity. Another limitation that plagues the runtime performance of serverless applications is the lack of a quality-of-service (QoS) control. As a result, functions suffer from the straggler issues [28]. *Therefore, an ideal serverless DAG framework should be able to provide effective workaround solutions.*

Opportunities: Running DAG parallel jobs (e.g., distributed linear algebra, distributed data analytics, etc.) has long been challenging for domain scientists and data analysts due to accessibility, configuration, provisioning, and cluster management complexity. The emerging serverless computing model seems to provide an attractive-enough foundation for potentially relieving the domain scientists and data analysts of the tedious cluster administration efforts. *However, to bridge the gap, current state-of-the-art badly requires a fast and efficient serverless-aware DAG framework middleware.*

B. Serverless Workflow Management Frameworks

Existing serverless frameworks have been built using two main approaches. The first is a queue-based master-worker approach in which the master orchestrates the workflow and submits tasks that are ready for execution to a queue. Workers are cloud functions that process these tasks in parallel when possible. For example, numpywren [29] is a serverless linear algebra framework. numpywren workers are implemented using PyWren [21], which is a framework for executing data-intensive batch processing workloads.

In the second approach, the master directly invokes cloud functions to process ready tasks [7, 3]. Examples of this include Sprocket [11] and ExCamera [15], which have been developed for serverless video processing. This second

---

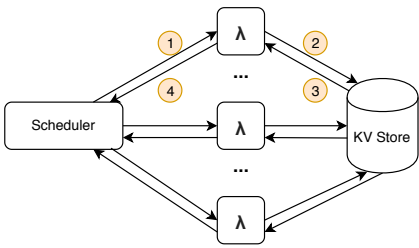[4]"Cold start" refers to the first-ever invocation of a function instance.

Figure 1: The strawman scheduler architecture. (1) The scheduler invokes a Lambda function, which establishes a TCP connection with the scheduler, executes the task, and then (2) sends the output results to the KV store. Once the Lambda function receives an ACK from the KV store (3), it notifies the scheduler (4) that the task is finished.
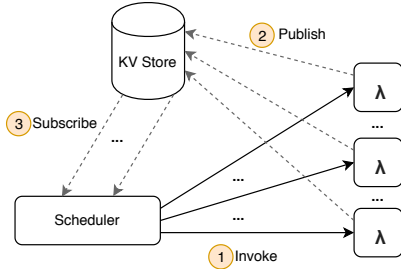

Figure 2: The pub/sub architecture. (1) The scheduler invokes a Lambda function, which executes the task, and then publishes the output results to the KV store in (2). The scheduler as the subscriber listens for messages on predefined channels, and gets notified in (3) whenever a Lambda function publishes the results to the KV store.
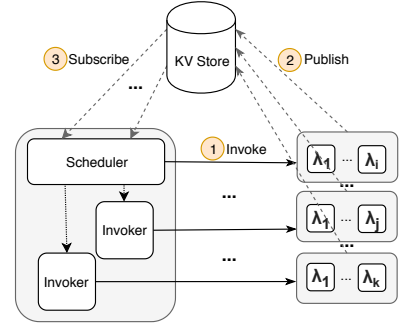

Figure 3: The parallel-invoker architecture. This architecture extends the pub/sub architecture in Figure 2 with a parallel-invoker that accelerates Lambda invocations by spawning multiple invoker processes in the scheduler to concurrently invoke Lambda functions.

approach is also used by general purpose serverless orchestration frameworks. Example frameworks include AWS Step Functions, Azure Durable Functions, Fission Workflows [6], and the framework in [24].

Lòpez et al. [23] evaluated AWS Step Functions and Azure Durable Functions with respect to their support for parallel execution, among other attributes. They found that overhead for parallelism grows exponentially with the number of parallel functions for AWS Step Functions and Azure Durable Functions.

Fission Workflows is built on top of the Fission [14] serverless framework for Kubernetes. Users define a DAG by creating a configuration file, which defines tasks and their dependencies. The framework in [24] is built upon the HyperFlow [7] workflow engine. HyperFlow models its workflows using user-written JSON files, and thus is similar to Fission Workflows with respect to how workflows are represented. While manually composing a DAG configuration may work well for coarse-grained microservice-based workflow applications, manually implementing a complex, fine-grained workflow is nontrivial. For this reason, [6, 24] are not well-suited for supporting complex computing jobs implemented using high-level programming languages.

WUKONG uses neither a master-worker-queue approach nor a direct invocation approach. Instead, WUKONG adopts a decentralized approach in which the global DAG is partitioned into local subgraphs. Each WUKONG executor is responsible for scheduling and executing the tasks within its assigned subgraph in an autonomous manner. An executor assumes the master's role when it uses its assigned subgraph to determine when its tasks can be executed; an executor assumes a worker's role when it executes these tasks. WUKONG executors coordinate to ensure that the dependencies in the global DAG are satisfied.

III. MOTIVATIONAL STUDY: A JOURNEY FROM THE SERVERFUL TO THE SERVERLESS

Prior to the emergence of the serverless computing model, DAG schedulers were designed to work with a finite number of compute and storage resources. These schedulers have to maintain a (complete or partial) global view of which tasks are running where, and use this view to optimize with respect to certain predefined objectives. Serverless computing, on the other hand, offers a nearly infinite amount of ephemeral resources, which are transparently managed by the service provider. Consequently, traditional schedulers would fail to utilize cloud resources optimally. WUKONG takes a radically different approach and is motivated by the above observations with the goal of improving the performance for task dispatching with respect to serverless platforms. In this section, we present our motivational study of designing a fast and efficient serverless DAG engine.

A. A Strawman Scheduler

We began our journey by implementing a centralized DAG scheduler, which simply parsed the user-defined job code, generated a DAG data structure, and sent off the DAG tasks to a group of Lambda functions for execution. Our strawman scheduler was a modification of the Python-written Dask distributed scheduler. Dask is an open-source parallel computing library for Python data analytics [4]. Traditionally, Dask distributed executes tasks within so-called worker processes, each running as a long-lived server across a cluster of machines. In Dask, the scheduler sends tasks to the worker processes for execution. Worker processes run as long-lived servers across a cluster of machines. The Dask distributed scheduler uses a communication protocol to communicate with workers and balance their load with respect to certain optimization constraints, such as

data locality and memory consumption. We reused the DAG and communication protocol modules from Dask, used AWS Lambda for task execution instead of worker processes, and disabled load balancing as load balancing is handled by AWS Lambda.

In a typical serverful distributed processing framework, worker processes can directly communicate with each other using TCP. A worker process that needs to execute a task `T` may find that `T`'s input data is not stored locally. This worker will then issue TCP requests for `T`'s input data to the workers who executed the upstream tasks that output this data. In our serverless computing environment, Lambda functions are not allowed to accept inbound TCP connection requests. Due to this constraint, the upstream tasks in WUKONG would have to store their output data in external distributed storage (a key-value store or KV store in short), from which the dependent downstream tasks can read their input data and make progress. Figure 1 depicts the strawman approach.

B. Publish/Subscribe Model

While the centralized strawman scheduler worked, it suffered from several performance bottlenecks. The first performance bottleneck was due to the large number of concurrent TCP connection requests sent to the scheduler from the Lambda functions. A short-lived Lambda function will immediately request a TCP connection with the scheduler to acknowledge the completion of its task. This makes it easy for a pool of thousands of newly invoked Lambda functions to overwhelm the scheduler. This is not a problem for a serverful deployment, e.g., a statically deployed Hadoop cluster with hundreds of worker nodes that established TCP connections at cluster initialization phase.

To address this problem, we adopted a pub/sub (publisher/subscriber) approach (Figure 2). The pub/sub scheduler provided higher performance than the strawman scheduler, since sending task completion messages through pub/sub channels was more efficient than using a large number of concurrent TCP connections; also, the number of network hops was reduced. The pub/sub architecture was easy to integrate, since external storage was already being used to store intermediate results.

C. +Parallel Invokers

While the pub/sub approach had substantially improved network performance, the framework struggled to launch Lambda functions quickly enough for large, bursty workloads. This is due to the large cost of invoking a Lambda function (e.g., invoking an AWS Lambda function takes about 50 milliseconds with the Boto3 AWS Python API). To scale-up Lambda invocation performance, we created a large number of dedicated Lambda-invoker processes co-located with the scheduler (Figure 3). When DAG task dependencies resolve, the scheduler evenly distributes task invocation responsibilities among multiple invoker processes, enabling (near-)linear speedup.
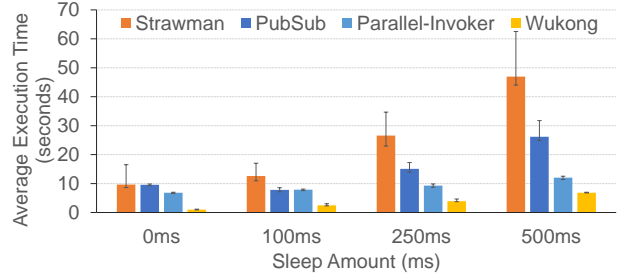

Figure 4: Performance comparison of different design iterations for Tree Reduction (TR). TR is a microbenchmark with a tree-like DAG topology [9], which combines neighboring elements until there is only one left. We ran TR with an initial array of 1024 numbers (i.e., 512 leaf tasks at the bottom of the DAG) on each system ten times and recorded the average (bars), and {min, max} (error bars). We intentionally added sleep-based delays in each task to simulate a compute task with a controllable duration.

Figure 4 plots the average execution time achieved by each design iteration. We intentionally added a sleep-based delay to each task in order to simulate a compute task with controllable duration. For TR test with 0ms sleep delays, the performance difference between the strawman and pub/sub versions of our framework is roughly the same due to the fact that the TR workload is primarily dominated by the communication overhead of transferring the array over the network. As noted above, the parallel-invoker version is able to execute TR 24.2% faster than strawman and pub/sub. This is because TR is also characterized by a large number of leaf tasks. Specifically, the TR algorithm generates $\frac{n}{2}$ leaf tasks, where n is the length of the input array. The parallel-invoker version can invoke the leaf tasks at a significantly higher rate than strawman and pub/sub; consequently, parallel-invoker performs better for workloads with a large number of leaf tasks. As the time span of each task increases, pub/sub start to show performance benefit against strawman, because a less number of TCP connections significantly reduces the amounts of IRQ requests which flood the strawman case. Parallel-invoker improves the performance against pub/sub, but is still sub-optimal due to network I/O overheads. The goal of WUKONG is to reduce the execution time of a DAG job, an optimal serverless DAG engine that dispatches DAG tasks with minimum runtime overhead.

One critical issue of the parallel-invoker architecture is its large commitment of resources to the centralized pub/sub scheduler throughout the whole course of the workload. Due to this, we moved to a decentralized scheduler design. This major design change came about as a result of a key observation, which was that the scheduler was only being used to launch downstream tasks as data dependencies of the DAG were resolved. Instead of scaling-up the invocation process of the centralized scheduler, each Lambda function could directly handle the responsibility of invoking downstream tasks without having to coordinate with the
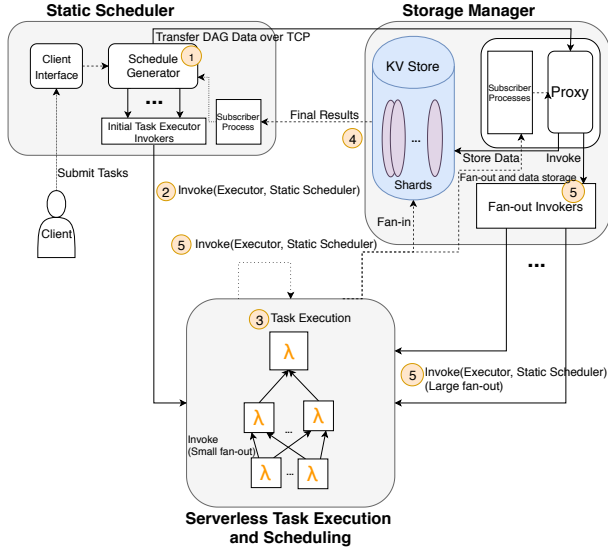
Figure 5: Overview of WUKONG architecture.

Figure 6: Static and dynamic scheduling.

centralized scheduler. This lead to an effective, serverless-aware, scale-out design that is described next.

IV. WUKONG DESIGN

In this section, we present the system design of WUKONG. We describe the high-level components and discuss the techniques used for static scheduling, task execution, dynamic scheduling, and managing storage.

A. High-Level Design

WUKONG consists of three major components: a static scheduler, a serverless task execution and scheduling runtime, and a storage manager. Figure 5 shows the high-level design of WUKONG. This figure reflects a major design revision to the Pub/Sub scheduler described at the end of §III. This revision removed the requirement for Lambda functions to acquire downstream tasks from the KV store. We modified the scheduler to produce static schedules, where each schedule represents a sub-graph of the DAG. A static schedule contains all the task code and other required (static) information, e.g., data dependencies, for each task in the sub-graph. The scheduler now passes a static schedule to each Lambda function it invokes, meaning that each function starts with all of the task code that it may have to execute. This removed the necessity for Lambda functions to grab downstream task code from the KV store, which speeds up execution by decentralizing WUKONG. Since static schedules contain the dependencies, scheduling operations for fan-in and fan-out processing can be done dynamically by the Lambda functions, which removes the need for a centralized scheduler to determine when data dependencies have been satisfied.

B. Static Scheduling

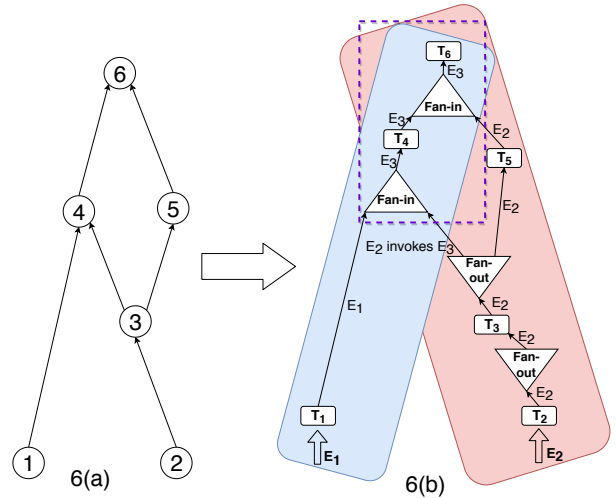WUKONG users submit a Python computing job to WUKONG's DAG generator, which converts the job into a DAG. The static Schedule Generator generates static schedules from the DAG. For a DAG with n leaf nodes, n static schedules are generated. A static schedule for leaf node L contains all of the task nodes that are reachable from L and all of the edges into and out of these nodes. The data for a task node includes the task's code and the KV Store keys for the task's input data. The schedule for L is easily computed using a depth-first search (DFS) that starts at L. Figure 6(a) shows a DAG with two leaf nodes. Figure 6(b) shows the two static schedules that are generated from the DAG — Schedule 1 is the nodes and edges in the region colored blue (left) and Schedule 2 is the nodes and edges in the region colored red (right).

Static schedules are used to reduce the number of network I/O operations that must be performed by the Task Executors, which improves overall system performance. Instead of executing a single task and retrieving the next task from the KV store, Task Executors receive a static schedule of all of the tasks (including the task code) they may possibly execute.

A static schedule contains three types of operations: task execution, fan-in, and fan-out. Note that there is at least one fan-in or fan-out operation between each pair of tasks. To simplify our description, when tasks T1 is followed immediately by task T2 in a DAG and T1 (T2) has no fan-out (fan-in), we add a trivial fan-out operation between T1 and T2 in the static schedule. This fan-out operation has one incoming edge from T1 and one outgoing edge to T2, i.e., there is no actual fan-out. In Figure 6(a) and Figure 6(b), this is the case for tasks T2 and T3.

A task operation may appear in more than one static schedule. In Figure 6(b), tasks T4 and T6 are both in Schedule 1 and Schedule 2. This will create a scheduling conflict between the Task Executors that are assigned to schedule these tasks, since tasks T4 and T6 should be executed only once. There is not enough information

available in the DAG to statically determine how to resolve scheduling conflicts so that execution time is minimized; instead, conflicts are resolved by dynamic scheduling operations performed by the Task Executors. Note also that a static schedule does not map a given task `T` to a processor; this mapping is done dynamically and automatically by the AWS Lambda runtime when the Task Executor that will (eventually) execute task `T` is invoked. A static schedule only specifies a valid partial-ordering of the tasks it contains — tasks are to be executed in bottom-up order, starting with the leaf node in the static schedule. Dynamic scheduling during task execution imposes the remaining constraints on task order. The time and place that tasks are executed is determined at runtime.

C. Task Execution and Dynamic Scheduling

Execution starts when the scheduler's Initial Task Executor Invokers assign each static schedule produced by the Schedule Generator to a separate AWS Lambda function, which we refer to as a *Task Executor*, and invoke the set of initial Task Executors in-parallel. Each of these Task Executors performs the first operation in its static schedule, which is always to execute the leaf node task in its static schedule. In Figure 6(b), Task executors `E1` and `E2` execute leaf tasks `T1` and `T2`, respectively. An Executor will then execute the tasks along a single path in its static schedule, enforcing the static ordering of tasks along the path.

If Task Executor `E` executes a fan-out operation with only one out edge, the operation has no effect — Executor `E` simply performs the next operation in its schedule, which will be to execute the next task. A Task Executor may thus execute a sequence of tasks before it reaches a fan-out operation with more than one out edge or a fan-in operation. For such a sequence of tasks, there is no communication required for making the output of the earlier tasks available to later tasks for input. All intermediate task outputs are cached in the local memory of the the Task Executor.

If Task Executor `E` executes a fan-out operation with n (where $n > 1$) out edges, then `E` invokes $n - 1$ new Task Executors. The intermediate output objects that are needed by the new Executors are sent to the Storage Manager for storage in the KV Store, and the associated keys are passed to the invoked Executors as arguments. Each of the $n - 1$ Executors will be assigned a static schedule that begins with one of the $n - 1$ out edges. Each of these (possibly overlapping) static schedules corresponds to a sub-graph of `E`'s static schedule. Executor `E` continues task execution and scheduling along the remaining out edge and executes the next operation encountered on this edge. We say that `E` becomes the Executor for one out edge and invokes Executors for the remaining $n-1$ out edges. In Figure 6(b), each fan-out operation has one edge labeled "becomes" and 0 or more out edges labeled "invokes". The label on an in or out edge also indicates the Executor that is performing the

dynamic scheduling operations that involve that edge. For Executor `E2`, the first fan-out operation is trivial. On `E2`'s second fan-out operation, `E2` becomes the Executor that will execute `T5` and invokes Executor `E3`.

As mentioned above, a fan-in operation represents a scheduling conflict between two or more Task Executors that are executing overlapping static schedules. If Task Executor `E` executes a fan-in operation with n (where $n > 1$) in-edges, then `E` and the $n - 1$ other Task Executors involved in this fan-in operation cooperate to see which one of them will continue their static schedules on the out edge of the fan-in. The Task Executors that do not continue will send their intermediate output objects to the Storage Manager and stop. In Figure 6(b), the first fan-in operation of Executors `E1` and `E3` resolves the conflict between their static schedules. We assume that `E3`'s fan-in operation is executed after `E1`'s fan-in operation; thus, `E1` stops and `E3` continues executing its static schedule at the out edge of the fan-out. At `E3`'s next fan-in operation, which also involves `E2`, we assume `E3` executes its fan-in operation last so that `E2` stops and `E3` executes task `T6` and then stops.

Task Executors cooperate on fan-in operations for a fan-in `F` by accessing an associated dependency counter for `F` that is stored in the KV Store. This counter tracks the number of `F`'s input dependencies that have been satisfied during execution. When a Task Executor `E` finishes the execution of a task that is one of the input dependencies of `F`, Executor `E` performs an atomic increment operation on the dependency counter of `F`. The updated value of the dependency counter is then compared against the number of input dependencies of `F`. If the value of the dependency counter is equal to the number of input dependencies, then all input dependencies of `F` have been satisfied and the task `T` on the out edge of `F` is ready for execution. In this case, task `E`, which executed the last dependent task of the fan-in, will continue its static schedule by executing `T`. If, instead, the value of the dependency counter is less than the number of input of `F`, then some input dependencies of `F` have yet to be satisfied. In that case, task `T` is not ready for execution, so `E` saves its intermediate output objects and stops. Notice that no Task Executor waits for any input dependencies of a fan-in to be satisfied. Note that AWS Lambda would bill Task Executors for wait time, which is why waiting is avoided.

For fault tolerance, we relied on the automatic retry mechanism of AWS Lambda, which allows for up to two automatic retries of failed function executions. In the future, we will investigate more advanced error handling mechanisms.

D. Storage Management

The Storage Manager performs various storage operations on behalf of the Task Executors and the Scheduler. At the start of workflow processing, the Storage Manager receives the workflow DAG and the static schedules derived from the DAG from the Scheduler.

Intermediate and Final Result Storage: Task Executors publish their intermediate and final task output objects to the KV Store. Final outputs are relayed to a Subscriber process in the Scheduler for presentation to the Client.

Small Fan-out Task Invocations: When a Task Executor performs a fan-out operation that has a small number of out edges, the Task Executor will make the necessary Executor invocations itself. However, sequentially executing a large number of invocations is time consuming so the Executor Task invocations are performed in parallel with assistance from the Storage Manager.

Large Fan-out Task Invocations: When a fan-out has a number of out edges that is larger than a user-specified threshold, the Task Executor publishes a message that is relayed to a Subscriber process in the Storage Manager, which passes the message to the Proxy. This message contains an ID that identifies the fan-out's location in the DAG. The Proxy uses the DAG and the fan-out ID to identify the fan-out's out edges in the DAG. This allows the Proxy, with the assistance of the Fan-out Invokers in the Storage Manager, to make the necessary Task Executor invocations, in parallel. The Proxy passes to each Executor its intermediate inputs (or their key values in the KV Store) and the Executor's static schedule.

V. PRELIMINARY RESULTS

We have implemented WUKONG using roughly $6,000$ lines of Python code (about $2,500$ LoC for the AWS Lambda Runtime, $2,400$ LoC for the Storage Manager, and $1,100$ LoC for the Static Scheduler). WUKONG currently supports AWS Lambda. Porting WUKONG to other public cloud and open source platforms is our work in progress.

Experimental Goals and Methodology. The goals of our preliminary evaluation were to:

- identify and describe the main factors influencing the performance and scalability of WUKONG,
- compare WUKONG against the serverful Dask framework to determine whether WUKONG can achieve comparable performance, even with the inherent limitations imposed by AWS Lambda.

We compare the performance of WUKONG against Dask `distributed` on two different setups: a five-node EC2 cluster with each virtual machine (VM) running five worker processes and a local setup on a laptop with four worker processes. We repeated the same tests on an easy-to-use laptop computer to further demonstrate that, with the same workload, WUKONG can achieve superior performance with minimal cluster administration effort.

Our evaluation was performed on AWS. The static scheduler ran in a `c5.18xlarge` EC2 VM and the KV Store was a Redis cluster partitioned across ten `c5.18xlarge` shards. The KV Store proxy was co-located on the same VM as one of the ten Redis shards. Each Lambda function was allocated 3GB memory with a timeout parameter set to two minutes.
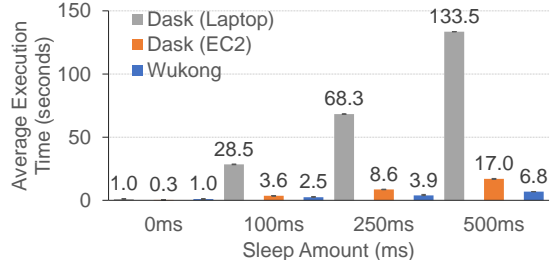

Figure 7: TR performance comparison.

Each node in the five-node cluster was an EC2 `t2.2xlarge` VM. We configured this cluster with general-purpose VMs to see if our serverless platform could match their performance. We opted to not configure a cluster of increased price and performance as we cannot configure our AWS Lambda functions to match the processing power of such a cluster. Further, we cannot control for the various restrictions held in place by Amazon including the rate at which we can invoke Lambda functions, the memory allocated to Lambda functions (above 3GB), the network resources allocated to each function, etc. The laptop was equipped with a two-core `Intel i5` CPU @ *2.30GHz*. Each Dask worker was allocated 2GB of laptop memory.

We describe the tested DAG applications as follows.

Tree Reduction (TR): TR sums the elements of an array. TR repeatedly adds adjacent elements until only a single element remains. The implementation used here is general-purpose; it is not optimized for a highly distributed, serverless algorithm, serving as a microbenchmark for effectively evaluating serverless DAG engine.

General Matrix Multiplication (GEMM): GEMM, as the core of many linear algebra algorithms, performs matrix multiplication. We evaluate the performance of WUKONG for GEMM with two different matrix sizes: $10,000 \times 10,000$ and $25,000 \times 25,000$.

Singular Value Decomposition (SVD): Two SVD workloads are used. The first workload computes the SVD of a tall-and-skinny matrix, i.e., a matrix with a significantly larger number of rows than columns (SVD1). The second workload computes the rank-5 SVD of an $n \times n$ matrix using an approximation algorithm (SVD2) provided by [18]. We use both SVD workloads as a real-world application to evaluate the performance of WUKONG for increasingly large SVD problem sizes.

Support Vector Classification (SVC): SVC is a real-world machine learning application. We evaluate the performance of WUKONG on SVC with increasingly large problem sizes. This workload is a benchmark that was retrieved from the publicly available Dask-ML benchmarks [5].

A. End-to-End Performance Comparison

As mentioned earlier, serverless computing suffers from cold starts. We address this issue by warming up a pool of Lambdas, which is the same strategy employed by Ex-Camera [15]. Due to AWS' planned cold-start performance
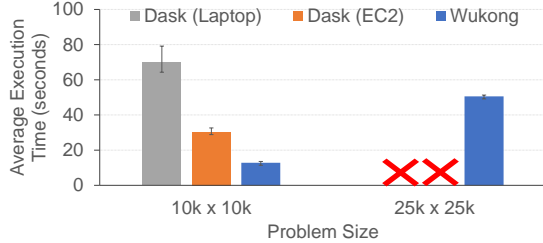
Figure 8: GEMM performance comparison.

Figure 9: SVD1: SVD of tall-and-skinny matrix.

Figure 10: SVD2: SVD of general matrix.

optimizations for Lambdas running within a virtual private cloud [25], "cold start" penalties should not be nearly as large of an issue in the future.

We first examine the performance of TR (for a preliminary analysis that compares the various design iterations please refer to Figure 4 in §III-C). As shown in Figure 7, WUKONG greatly outperforms all previous versions of the framework. The decentralized scheduler reduces the network I/Os required to complete the workload; however, due to the extremely short-duration add operations used by TR with 0ms sleep delays, the communication overhead of transferring the underlying array greatly outweighs the performance gains from increased parallelism. This is why WUKONG achieves lower performance than Dask (EC2). WUKONG outperforms all other execution platforms when small sleep delays are added to each operation of the tree reduction. WUKONG executes $2.5\times$ faster than Dask (EC2) in the case of 500ms delays. These small delays simulate additional work for each task. The results of this workload with added delays indicate that for workloads with longer tasks, the increased parallelism provided by WUKONG outweighs the communication overhead, demonstrating that our decentralized DAG scheduler incurs minimum overheads.

The results of our GEMM tests further demonstrate WUKONG's superiority in elasticity and performance. In the case of case of $10,000 \times 10,000$ matrix multiplication, WUKONG executed the workload more than twice as fast as Dask (EC2) and more than five times as fast as Dask (Laptop). Dask (EC2) could likely perform this workload faster if the cluster was larger, whereas for WUKONG, it leverages the large number of CPUs provided by AWS Lambda to elastically scale up the performance. When multiplying $50,000 \times 50,000$ matrices, both setups of Dask (Laptop and EC2) suffered from out-of-memory (OOM) errors, failing to complete the job. Our analysis of GEMM on WUKONG indicates that these workloads were dominated by the communication overhead of transferring portions of the matrix to the Task Executors.

Next we analyze the performance of the two SVD workloads. For SVD1, we used the following numbers of rows: $200k$, $400k$, $800k$, and $1,000k$. Figure 9 shows that both Dask (EC2) and WUKONG were able to greatly outperform Dask (Laptop). For the first two problem sizes, Dask (EC2) out-performed WUKONG; however, as the problem size 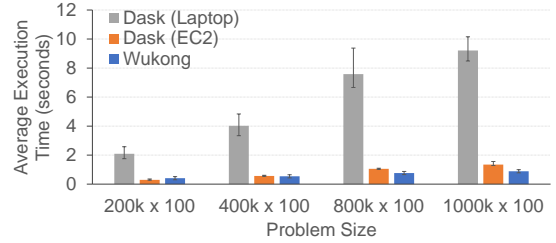increased, the performance of WUKONG began to exce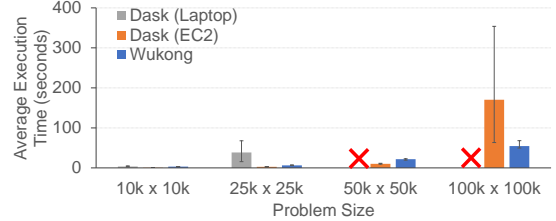ed that of Dask (EC2). This is because the parallelism from AWS Lambda began to outweigh the communication overhead of the workload. Even so, the overhead associated with network I/Os was a significant factor in the performance of this workload on WUKONG.

The dominance of communication in SVD was further demonstrated by the first three workload sizes of the SVD2 on a general $n \times n$ matrix (Figure 10). Dask (EC2) was faster than WUKONG for relatively smaller problem sizes, since the statically-deployed Dask `distributed` cluster supports direct worker-to-worker communication with less network I/O overhead (especially for large intermediate results), and the CPU resources of the cluster did not yet become a bottleneck. Additionally, Dask (Laptop) suffered from OOM errors in the $50k \times 50k$ case and was unable to complete the workload. Finally, WUKONG executed the $100k \times 100k$ workload $3.1\times$ faster than Dask (EC2), again because of the elasticity of WUKONG. WUKONG does not require extra administration effort for scaling out the computation capacity, whereas Dask (EC2) would do, thus imposing extra burden to the end users. The number of Lambda functions used for each of the workloads was 84, 480, 295, and 1082, respectively. The $50k \times 50k$ workload used less Lambdas than the $25k \times 25k$ workload due to the strategy used to partition the initial input data. Different input data partitioning strategies may introduce different parallelism-communication tradeoffs and affect scalability. We plan to investigate partitioning strategies as part of our future work.

Finally, we analyze the performance of SVC on WUKONG (Figure 11). We varied the SVC problem size (in this case, number of samples) over the values $100k$, $200k$, $400k$, and $800k$. While Dask (EC2) completes the job slightly faster than WUKONG for the smallest problem size, the performance of WUKONG begins to exceed Dask (EC2) as the problem size increases. The performance gap increases as the problem size varied from $400k$ to $800k$. For a sample number of $800k$, WUKONG is able to execute the workload
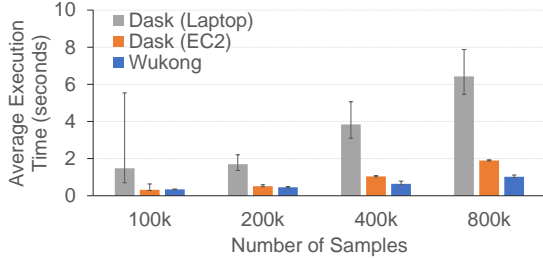
Figure 11: Performance comparison of SVC machine learning classification.
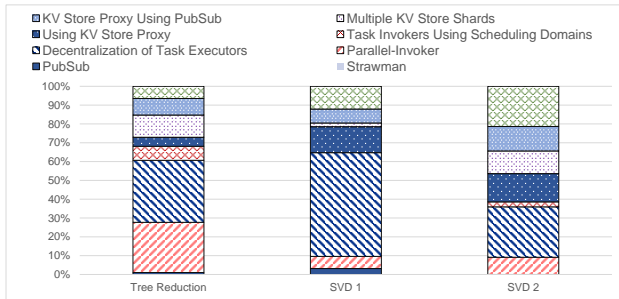

Figure 12: Contributing factors of different optimization techniques employed in WUKONG.

nearly $2\times$ as fast as Dask (EC2). This again strengthens our confidence that WUKONG can serve as a generic DAG engine for accelerating complex real-world applications such as machine learning.

B. Factor Analysis

WUKONG is able to effectively scale out to support large problem sizes and workloads. Figure 12 shows the amount that each major version of WUKONG contributed to the overall performance improvement from the original *Strawman* version to the current version. The most significant improvement came as a result of the decentralization of the Task Executors. Prior to Task Executor decentralization, Task Executors would only execute the task initially given to them by the static scheduler. Once decentralized, Task Executors instead retrieved new tasks from the KV Store each time they completed the execution of their current task.

Other significant improvements to the overall performance of WUKONG included the use of dedicated task invoker processes, which originated in the *Parallel Invokers* version, and the use of the KV Store Proxy to parallelize large task fan-outs. The effect of the KV Store Proxy varied depending on the workload since workloads that lacked high fan-outs would not actually utilize the proxy. Switching the communication protocol used by the KV Store Proxy from TCP to Redis PubSub also resulted in a fairly substantial performance improvements. Just as for the static scheduler, Redis PubSub enabled the KV Store Proxy to handle a higher volume of messages from Task Executors. Finally, running each KV Store shard on its own separate VM resulted in a significant performance improvement. Initially, all KV Store shards were running on the same VM, which
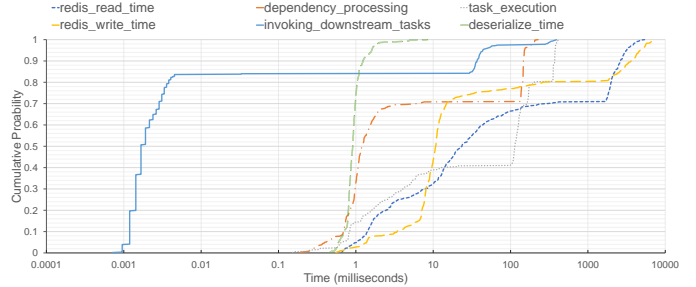

Figure 13: CDF breakdown of tasks in SVD2 with a $50k \times 50k$ matrix.

resulted in resource contention for network bandwidth. Placing each shard on its own VM eliminated this bottleneck.

C. Overhead Quantification

The overhead associated with storing and retrieving large intermediate data values during workload execution is a major factor that impacts WUKONG's performance. For workloads characterized by short tasks and large communication overheads, WUKONG is not able to outperform Dask (EC2). This is most prevalent in the tree reduction workload without sleep delays, shown in Figure 7, and when computing the SVD of a square matrix, shown in Figure 10.

To quantify such I/O overhead, we conduct a detailed analysis with SVD2, by breaking down WUKONG's execution duration into fine-grained factors. Figure 13 shows a latency distribution of individual tasks in SVD2 of a $50k \times 50k$ matrix. We observe that there were a small number of KV store read and write operations which took upwards of ten seconds to complete. While a majority of tasks did not experience such communication overhead, the long network I/Os experienced by a minority of the tasks have a large impact on the workload's overall performance.

In order to estimate the improvement in performance that WUKONG could obtain if we were to use an ideally-fast (i.e., fully-optimized) intermediate storage, we executed a modified variant of SVD2 in which all array data was randomly generated each time it was used (instead of being written in and retrieved from the KV store). In Figure 10, the right-most (yellow-colored) bar shows the performance of WUKONG with this ideal intermediate storage. While the performance of Dask (EC2) is still better than WUKONG (with idea storage) for the smallest workload size, the performance is roughly the same in the $25k \times 25k$ case. Moreover, WUKONG (with ideal storage) is able to perform $1.67\times$ faster than Dask (EC2) for the $50k \times 50k$ workload in this experiment. As discussed earlier, WUKONG (in its current form) is already able to outperform Dask (EC2) by over 115 seconds on-average for the largest problem size. When using an ideal KV store, WUKONG would execute the workload in 95.50% less time than Dask (EC2). These results highlight the magnitude by which network communication overhead negatively affects the overall performance of WUKONG.

Data locality is another key factor which influences the performance of WUKONG. Increased data locality enables Task Executors to carry out the workload without needing to retrieve dependent inputs from the KV Store. This reduces the communication overhead, thereby increasing the performance of the framework. The overall effect of data locality largely depends on the size of the data values kept in the Task Executor's local storage. Our analysis of the network I/O performance found that the transfer of intermediate data objects that were tens to hundreds of megabytes in size were the cause of longer execution times (as opposed to smaller intermediate data objects). Consequently, WUKONG is able to utilize the local data stores on Task Executors most effectively when large objects are stored.

D. Limitations

One limitation of our evaluation is that we did not compare WUKONG against other serverless DAG engines. This was because all of the systems use different representations for their DAG's, and these representations are large and complicated. Consequently, it is nontrivial to convert a DAG from one system to another. A comparison between serverful Dask and WUKONG is possible because they use the same DAG representation. We are currently investigating DAG representations of other serverless DAG engines so that we can make a thorough comparison between WUKONG and other frameworks and understand the pros and cons of their design decisions.

VI. Conclusion

We have presented WUKONG, a high-performance DAG engine that implements decentralized scheduling by exploiting the elasticity and scalability of AWS Lambda to reduce network I/O overhead and improve data locality. Our evaluation shows WUKONG is competitive with a traditional serverful DAG scheduler Dask and demonstrates that decentralizing task scheduling contributes a significant portion of the improvement in overall performance. As part of our future work, we are exploring new techniques to fundamentally improve the performance of intermediate storage for serverless DAG workloads.

WUKONG is open sourced and is available at: https://github.com/mason-leap-lab/Wukong.

References

[1] 2018 Serverless Community Survey: huge growth in serverless usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

[2] Alibaba Cluster Trace Program (New 2018 Version. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.

[3] AWS Step Functions. https://aws.amazon.com/step-functions/.

[4] Dask: Scalable Analytics in Python. https://dask.org/.

[5] Dask: Scalable Machine Learning in Python. https://dask-ml.readthedocs.io/en/latest/#.

[6] Fission Workflows. https://github.com/fission/fission-workflows.

[7] HyperFlow: a scientific workflow execution engine. https://github.com/hyperflow-wms/hyperflow.

[8] Serverless: Build and run applications without thinking about servers. https://aws.amazon.com/serverless/.

[9] Tree Reduction benchmark. https://matthewrocklin.com/blog/work/2017/07/03/scaling.

[10] AKKUS, I. E., CHEN, R., RIMAC, I., STEIN, M., SATZKE, K., BECK, A., ADITYA, P., AND HILT, V. SAND: Towards high-performance serverless computing. In *USENIX ATC 18*.

[11] AO, L., IZHIKEVICH, L., VOELKER, G. M., AND PORTER, G. Sprocket: A serverless video processing framework. In *ACM SoCC '18*.

[12] CHENG, Y., CHAI, Z., AND ANWAR, A. Characterizing co-located datacenter workloads: An alibaba case study. In *ACM APSys '18*.

[13] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[14] FISSION. Serverless functions for kubernetes. https://fission.io/.

[15] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *USENIX NSDI 17*.

[16] GRAY, J. Why do computers stop and what can be done about it?, 1985.

[17] GUO, J., CHANG, Z., WANG, S., DING, H., FENG, Y., MAO, L., AND BAO, Y. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *ACm IWQoS '19*.

[18] HALKO, N., MARTINSSON, P.-G., AND TROPP, J. A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions.

[19] HELLERSTEIN, J. M., FALEIRO, J. M., GONZALEZ, J. E., SCHLEIER-SMITH, J., SREEKANTI, V., TUMANOV, A., AND WU, C. Serverless computing: One step forward, two steps back.

[20] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *USENIX HotCloud 16*.

[21] JONAS, E., PU, Q., VENKATARAMAN, S., STOICA, I., AND RECHT, B. Occupy the cloud: Distributed computing for the 99%. In *ACM SoCC '17*.

[22] JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C.-C., KHANDELWAL, A., PU, Q., SHANKAR, V., MENEZES CARREIRA, J., KRAUTH, K., YADWADKAR, N., GONZALEZ, J., POPA, R. A., STOICA, I., AND PATTERSON, D. A. Cloud programming simplified: A berkeley view on serverless computing. Tech. rep., 2019.

[23] LPEZ, P. G., SNCHEZ-ARTIGAS, M., PARS, G., PONS, D. B., OLLOBARREN, . R., AND PINTO, D. A. Comparison of FaaS orchestration systems. 148–153.

[24] MALAWSKI, M., GAJEK, A., ZIMA, A., BALIS, B., AND FIGIELA, K. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems* (Nov. 2017).

[25] MUNNS, C. Announcing improved VPC networking for AWS Lambda functions.

[26] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *USENIX ATC 18*.

[27] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, low latency scheduling. In *ACM SOSP '13*.

[28] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *USENIX NSDI 19*.

[29] SHANKAR, V., KRAUTH, K., PU, Q., JONAS, E., VENKATARAMAN, S., STOICA, I., RECHT, B., AND RAGAN-KELLEY, J. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[30] WANG, L., LI, M., ZHANG, Y., RISTENPART, T., AND SWIFT, M. Peeking behind the curtains of serverless platforms. In *USENIX ATC 18*.

[31] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI 12*.